Longest path problem in a DAG

Alex Gabriel Fraga da Silva Rodrigo Quednau Sandler

June 14 2024

Abstract

The following document presents a detailed report on the steps taken to solve the proposed problem in the second assignment of the Algorithms and Data Structures II class, which is to find the longest possible nesting sequence given a text file filled with dimensions of boxes. The document demonstrates the reasoning for approaching the problem, provides a detailed description of the developed solution, and presents results from multiple test cases, including algorithm run-times. The algorithm's time complexity, which has been determined to be $O(n^{1.22})$, is also analyzed.

1 Introduction

The problem´s scenario was to develop an algorithm designed to nest boxes one inside another as many times as possible, in order to find the biggest nesting sequence. A catalog with information of boxes is provided as a text file, where each line contains three integers ranging from 1 to 999, representing the dimensions of a box, but not identifying its height, width or length. The goal was to read and process the data from this file, determining the wished sequence.

With that settled, the first steps towards solving the problem were to abstract the challenge, select the programming language to be used and choose the most appropriated data structure to handle the problem. Already at first glance, we observed that using a graph could be very helpful, since the challenge's characteristics fit just fine with graph structures, more precisely a D.A.G. - directed acyclic graph. It will be built with vertices being boxes and the edges meaning a box can fit into another, exactly like exemplified in the assignment's description.

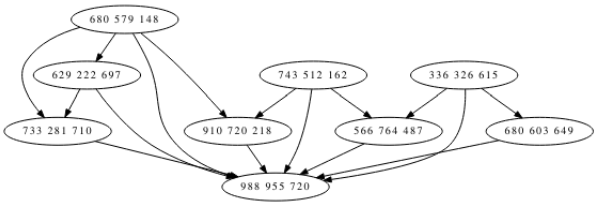

Figure 1: Graph example in the assignment's description.

2 Development

Once decided to use a graph, a programming language had to be picked and our choice was `C`. The longest path problem is known to have a Big O time complexity of $O(n^2)$, and C is known to be very fast, so we judged it to be a great match for a optimized solution.

As the proposal of the work was to be done in `Java`, we checked with the institution whether we could develop it in `C` or not. After the professor confirmed the request, we began the process of brainstorming the solution steps and defining the basics structs, which would be the pillar of the problem solving.

Looking at the test cases provided by the class' teacher, we quickly identified that a very large amount of boxes would be available in each catalog, and the graphs would be sparse. With these informations, the go to graph implementation was the adjacency list, once the adjacency matrix would be very expensive memorywise, and its better use is for non-sparse graphs.

2.1 Base structs

The box struct contains the three measures of a box, as shown here:

```
struct box:
   its lowest measure
   its middle measure
   its biggest measure
```

After that we also created structs for vertices in the graph, which contains the following components:

```
struct vertex:
    struct box
    struct edge pointer
    integer index
```

Finally, we created a struct for edges, described here:

```
struct edge:
   struct vertex pointer
   struct edge pointer
```

2.2 File reading and graph initialization

As stated beforehand, the test cases are formatted files in a way such each line represents the measures of a box. The values themselves do not specify which one is the height, width and length, but that is not a problem, since

we can rotate a box, making all of its dimension labels change, nulling the importance of labeling the measures.

So, we created the method `create_vertices(filename, count)`, which initialize the graph, creating vertices while reading each line. The method's parameters are the name of a catalog file and a count variable, so we can first find the number of boxes to be created, and after that, properly dynamically allocate memory for the graph structure.

The following is a pseudo code for the discussed function.

Algorithm 1 Getting number of boxes

```
1: file ← open file
2: if file not null then
3:    print "Could not open file: filename"
4:    return null
5: end if
6: ch
7: count ← 1
8: while not end-of-file do
9:    ch ← line
10:   if ch is equal to '\n' then
11:      count ← count + 1
12:   end if
13: end while
14: rewind file to the begin
15: return file
```

Algorithm 2 Creating the boxes

```
1: for i ← 0 to (count − 1) do
2:    read(integerA integerB integerC)
3:    vertices[i].b.lowest ← integerA
4:    vertices[i].b.middle ← integerB
5:    vertices[i].b.biggest ← integerC
6:    sort_box(vertices[i].b)
7:    vertices[i].head ← null
8: end for
9: close file
```

Past that, with all vertices already created, we follow to the edges creation.

2.3 Edges criation

With the basics established, we needed to be able to verify whether a box `A` fits into another box `B`, and we concluded that for a box `A` to fit in a box `B`, all of `A`'s measures should be lower than each of `B`'s measures in order, so:

- `A`'s lowest measure has to be smaller than `B`'s lowest measure
- `A`'s middle measure has to be smaller than `B`'s middle measure
- `A`'s biggest measure has to be smaller than `B`'s biggest measure

Once we decided the way to verify the boxes' fitting relationship, we can now build the function that establishes it. The adjacency list was created in a way that every vertex has its own linked list of adjacency, meaning: if `A` fits inside `B`, `A` is a node in `B`'s linked list of adjacency, and so on.

For that, it was initially created two loops, which compared each vertex array's position with all the others. But for optimizing purposes, we opted to, before making the comparisons, sort the array by volume. The decision of using the volume was made because, not necessarily a box with bigger volume will store a smallest one, but, if a box fits into another, the second one will certainly have a bigger volume. Then it was possible to have a faster and more efficient way of finding the edges, so, now, the loops can verify the boxes' relationships only for the subsequent positions.

With that, decreasing for a $n \cdot n$ number of comparisons to a total of

$$\left(\frac{n \cdot (n+1)}{2}\right) - n$$

comparisons. This number does not improve the polynomial complexity, but it does improve the average case scenario by a lot.

2.4 Longest nesting

Initially, we considered using Breadth-First Search combined with a stack to topologically sort the vertices array for solving the longest nesting problem. However, we soon realized that since we had already sorted the array by box volume, topological sorting was unnecessary. Sorting by volume ensures that the index of any given box is always lower than the index of any box it can fit into. Topological sorting is typically employed in dynamic programming to solve problems like finding the longest path, ensuring all sub-problems are resolved before tackling the current problem. In our scenario, once we identify a box, we've already determined the longest paths for all boxes that can fit into it. Therefore, we simply select the longest of these paths and add one to determine the longest path for the current box.

After the previous realization, we created a max depth array, which have the same size of the vertices array, with all the positions initialized to 1, as if a box fits into none, its longest nesting sequence is itself. The search for the longest nesting is described in the following pseudo code.

Algorithm 3 Calculate Maximum Depth

```
1: maxDepth ← allocate memory for an array of size g.size
2: for i ← 0 to g.size − 1 do
3:     maxDepth[i] ← 1
4: end for
5: for i ← 0 to g.size − 1 do
6:     e ← g.vertices[i].head
7:     while e is not null do
8:         u ← e.dest.index
9:         if maxDepth[u] < maxDepth[i] + 1 then
10:            maxDepth[u] ← maxDepth[i] + 1
11:        end if
12:        e ← e.next
13:    end while
14: end for
15: max ← 0
16: for i ← 0 to g.size − 1 do
17:    if maxDepth[i] > max then
18:        max ← maxDepth[i]
19:    end if
20: end for
21: return max
```

3 Results

Using the solution presented above, we ran multiple test cases and obtained the following results:

Text case	Number of boxes	Longest nesting	Runtime (milliseconds)
caso00010.txt	10	3	0.175
caso00020.txt	20	5	0.188
caso00050.txt	50	8	0.420
caso00100.txt	100	12	0.689
caso00200.txt	200	17	0.672
caso00300.txt	300	20	2.12
caso00500.txt	500	24	7.418
caso01000.txt	1,000	32	23.434
caso02000.txt	2,000	42	63.948
caso05000.txt	5,000	59	233.776
caso10000.txt	10,000	76	819.919

Table 1: Table relating the test cases, number of boxes, longest nesting and runtime.

It is important to say that the results were achieved by running the solution

in a MacBook Air 13 with Apple M1 chip and 8 GB of memory. The execution in other machines can provide results in different run-times.

4 Algorithm analysis

To analyze the algorithm, we only considered the Big O notation for the solution and not the graph creating, which besides the creating edges - that we already discussed - have a linear time complexity. The:

```
findLongestNesting(graph *g)
```

- which its pseudo code was presented at Section 2.4 - is the method that does the longest path finding process and its analysis follow:

This analysis is made applying techniques taken from the professor Marcelo Cohen's material [1].

For it, we used the same run-times from the results table, and plotted graphics using `Python`'s math.plot library.

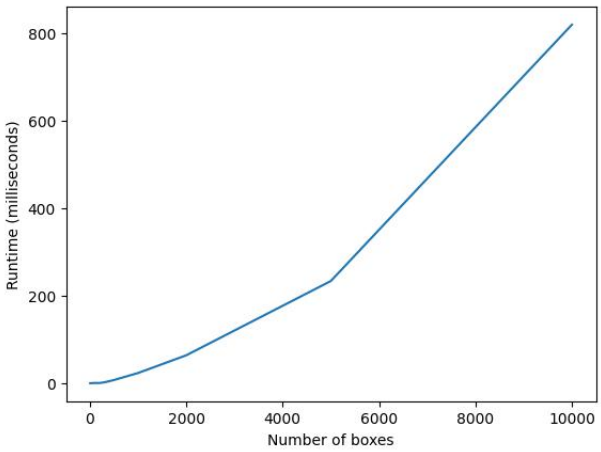

Figure 2: Graphic representing the runtime in function of each test case number of boxes.

The presented results in Figure 2 confirms the expected behavior for the polynomial complexity previously hypothesized. But, we need to handle the result with the right technique. That is why we used the professor's material, which defines the following rules:

- If the graph is an upward curvature and, after converting the y-axis to logarithmic scale, it becomes a straight line, the complexity is exponential.
- If case 1 did not result in a straight line, also convert the x-axis to logarithmic scale. If it results in a linear graph, then the complexity is polynomial.

- If neither of the two previous conditions are satisfied, then it cannot be asserted with certainty whether the complexity is exponential, polynomial or none of those.

The first rule was applied, which returned the following result:

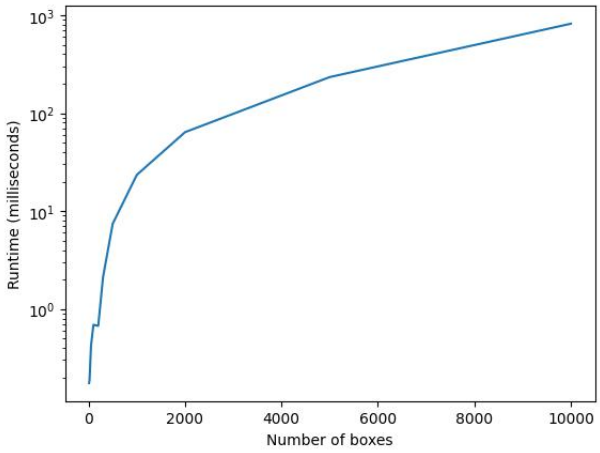

Figure 3: Graphic representing the runtime in function of each test case number of boxes, built with the y-axis in logarithmic scale.

As seen, the algorithm does not configure as exponential, so, the second rule was tested in order to verify the possibility of being polynomial.

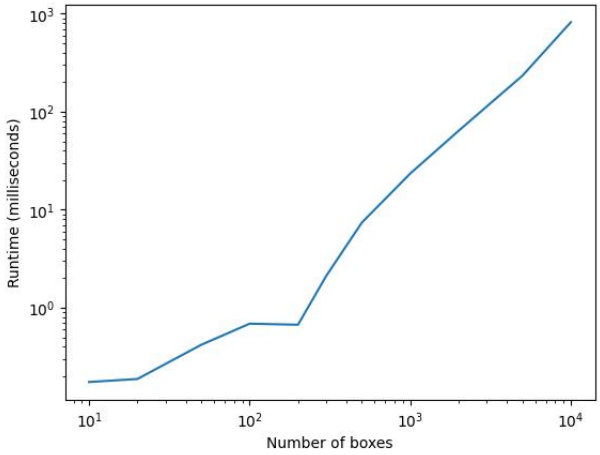

Figure 4: Graphic representing the runtime in function of each test case number of boxes, built with both y and x axis in logarithmic scale.

The reason why the line is not perfectly linear is due to the use of execution time on the y-axis, which can show fluctuations due to each machine's operation management and system. Despite that, it can still be characterized as such because it follows a linear trend, especially if we ignore the beginning of the line, which is allowed.

For better visualization of its linearity, we constructed another graph, this time drawing a parallel linear line from the point where it exhibits this behavior. The result can be seen below.

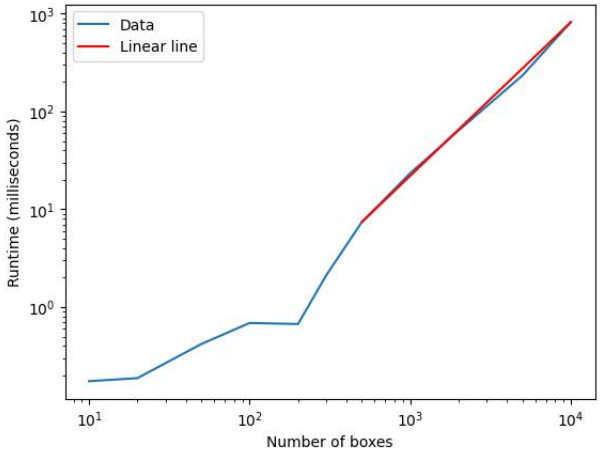

Figure 5: Graphic presenting the runtime in function of each test case number of boxes, built with both y and x axis in logarithmic scale and a linear parallel line.

4.1 Exponent calculus

Furthermore, it is essential to discover the exponent to which n is raised. For this purpose, we also utilized Professor Cohen's material.

In our quest to find the exponent, we will consider the formula below

$$f(n) = n^b$$

So, it is applied logarithm in both sides.

$$\log(f(n)) = \log(n^b) = b \cdot \log(n)$$

But, we will use $f(e^n)$ instead of $f(n)$ because, this way, we can eliminate the logarithms on the right side, like this:

$$\log(f(e^n)) = \log((e^n)^b) = b \cdot \log(e^n) = n \cdot b \cdot \log(e) = n \cdot b \cdot 1$$

Thus, we establish that b is the slope coefficient of this line. Now, we can calculate the value of the function for two known points. In this case, we will use the test cases of 10 and 10,000 boxes, as shown below:

$$f(10) = 0.175$$

$$f(10,000) = 819.919$$

As we are using logarithmic scale on both axes, we calculate b using the logarithm in both the numerator and denominator. By doing this, we obtain:

$$b \approx \frac{\log(819.919) - \log(0.175)}{\log(10,000) - \log(10)} = 1.2235776339\ldots$$

$$f(n) \approx n^{1.22}$$

Therefore, as demonstrated through two observations, the solution has polynomial complexity, denoted by $\mathrm{O}(n^{1.22})$.

5 Conclusion

The solution presented effectively meets the challenge with an optimized average-case runtime, demonstrating efficient performance within its polynomial complexity.

For further work, it would be interesting to implement methods that encompass general cases or are formatted differently. It would also be nice to build a graphical interface to simulate the context of a website and visually verify the chosen catalogs.

References

[1] Cohen, M. Análise de complexidade de funções genéricas. Disponível em: `https://colab.research.google.com/drive/1EU0uQuWkkDgGksDWb-5ZBQ6aRhCNB1bz?usp=sharing`. Acesso em: 24 de junho de 2024.